UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

(2)

AD-A205 339

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: InterAct Corporation, InterAct Ada 1750A Compiler System, Release 3.0 VAX 11/785 (Host) to Fairchild F9450/1750A (Target)	5. TYPE OF REPORT & PERIOD COVERED 15 July 1988 to 15 July 1989
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) National Bureau of Standards Gaithersburg, MD	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS National Bureau of Standards Gaithersubrg, MD	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE
	13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) National Bureau of Standards Gaithersuburg, MD	15. SECURITY CLASS (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 If different from Report)

UNCLASSIFIED

DTIC ELECTE FEB 0 7 1988 S D H

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

InterACT Ada 1750A Compiler System, Release 3.0, InterACT Corporation, National Bureau of Standards, VAX 11/785, VMS, Version 4.5 (Host) to Fairchild F9450/1750A, none, none, ACVC 1.9

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

89 2 6 044

AVF Control Number: NBS88VACT520

Ada Compiler
VALIDATION SUMMARY REPORT:
Certificate Number: 880715S1.09153
InterACT Corporation
InterACT Ada 1750A Compiler System, Release 3.0
VAX 11/785 Host, Fairchild F9450/1750A Target

Completion of On-Site Testing:
15 July 1988

Prepared By:
Software Standards Validation Group
Institute for Computer Sciences and Technology
National Bureau of Standards
Building 225, Room A266
Gaithersburg, Maryland 20899

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: InterACT Ada 1750A Compiler System, Release 3.0

Certificate Number: 880715S1.09153

Host:	Target:
VAX 11/785	Fairchild F9450/1750A
VMS,	none,
Version 4.5	none

Testing Completed 15 July 1988 Using ACVC 1.9

This report has been reviewed and is approved.

Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Bureau of Standards
Gaithersburg, MD 20899

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria, VA 22311

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies—for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

This information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of test are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

To attempt to identify any unsupported language constructs required by the Ada Standard

To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by the National Bureau of Standards according to policies and procedures established by the Ada Validation Organization (AVO). On-site testing was completed on 16 July 1988 at InterACT Corporation, New York, New York.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Software Standards Validation Group
Institute for Computer Sciences and Technology
National Bureau of Standards
Building 225, Room A266
Gaithersburg, Maryland 20899

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines. Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide., December 1986.

1.4 DEFINITION OF TERMS

ACVC — The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.

Ada Commentary — An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.

Ada Standard — ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant — The agency requesting validation.

AVF — The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.

AVO — The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.

Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Language Maintenance	The Language Maintenance Panel (LMP) is a committee established by the Ada Board to recommend interpretations and Panel possible changes to the ANSI/MIL-STD for Ada.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	An Ada program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A

test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of

REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: InterACT Ada 1750A Compiler System, Release 3.0

ACVC Version: 1.9

Certificate Number:	880715S1.09153
Host Computer:	
Machine:	VAX 11/785
Operating System:	VMS Version 4.5
Memory Size:	16 MB
Target Computer:	
Machine:	Fairchild F9450/1750A
Operating System:	none
Memory Size:	64 KB
Communications Network:	VAX/64000 Interface Software

The A.C.T. Ada compiler and linker run on VAX/VMS and produce 1750A load module files on the VAX. These load modules are in ACT 1750A Linker format. An ACT proprietary tool, ADA_H, is then run on the VAX to produce load modules files in Hewlet Packard (HP) 64000 format. HP's VAX/64000 interface software is then used to transfer the load module to the HP 64000 Workstation, containing the 1750A chip (a Fairchild 9450), run the load module on the 1750A processor, and then transfer output from the run back to the host VAX. This transfer-run-transfer sequence is entirely under VAX/VMS control and requires no manual intervention at

the workstation. The output produced during a run is created using 64000 simulated disk I/O. A HP 64286A Emulation Probe with a 64271/AB control board is used to house the 1750A chip. This unit is attached to the HP 64000 Workstation.

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 10 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See test D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

  This implementation supports the additional predefined types LONG_INTEGER and LONG_FLOAT in the package STANDARD. (See tests B86001BC and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Expression evaluation.

  Apparently all default initialization expressions or record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

  Assignments for subtypes are performed with less precision than the base type. (See test C35712B.)

  This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

  Apparently NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

  Apparently NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

  Apparently underflow is not gradual. (See tests C45524A..Z.)

- Rounding.

  The method used for rounding to integer is apparently round away from zero. (See tests C46012A..Z.)

  The method used for rounding to longest integer is apparently round away from zero. (See tests C46012A..Z.)

  The method used for rounding to integer in static universal real expressions is apparently round away from zero. (See test C4A014A.)

- Array types.

  An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

  NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

  NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

  A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array objects are declared. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array subtypes are declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications indications during compilation. (See test E38104A.)

In assigning record types with disciminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Aggregates.

In the evaluation of a multi-dimensional aggregate, index subtype checks appear to be made as choices are evaluated. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

- Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is not supported, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are not supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are not supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

Record representation clauses are supported, however the alignment clause is not supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are not supported. (See test C87B62A.)

- Pragmas.

The pragma INLINE is not supported for procedures. The pragma INLINE is not supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

The director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE_ERROR or NAME_ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO.

- Generics.

Generic subprogram declarations and bodies can compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

Generic ~~units~~ bodies must be compiled before their ~~bodies~~ ~~units~~ units are instantiated.

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tests, 28 tests had been withdrawn because of test errors. The AVF determined that 507 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing. Modifications to the code, processing, or grading for 10 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	107	1048	1360	16	11	45	2587
Inapplicable	3	3	493	1	6	1	507
Withdrawn	3	2	21	0	2	0	28
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	184	462	490	245	164	98	139	326	132	36	233	3	75	2587
Inapplicable	20	110	184	3	1	0	4	1	5	0	1	0	178	507
Withdrawn	2	14	3	0	1	1	2	0	0	0	2	1	2	28
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 28 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C	C35904A
C35904B	C35A03E	C35A03R	C37213H	C37213J	C37215C
C37215E	C37215G	C37215H	C38102C	C41402A	C45332A
C45614C	E66001D	A74106C	C85018B	C87B04B	CC1311B
BC3105A	AD1A01A	CE2401H	CE3208A		

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 507 test were inapplicable for the reasons indicated:

C35508I..J (2 tests) and C35508M..N (2 tests) use enumeration representation clauses for derived types which are not supported by this compiler.

C35702A uses SHORT_FLOAT which is not supported by this implementation.

C35A06N compiled code exceeds the 64K memory capability of the target.

C36003A type declaration exceeds the capability of the compiler.

A39005B and C87B62A use length clauses with SIZE specifications which are not supported by this compiler.

C87B62B defines an access type's collection size using a length clause, where the length clause value is the collection size of another access type that does not have a collection size length clause. The compiler defines collection size in the latter case as arbitrarily large; as a consequence, the attempt to use it in a collection size length clause raises STORAGE_ERROR, as an arbitrarily large object cannot be allocated by the compiler (the limit is 32K words).

A39005G uses an alignment clause which is not supported by this compiler.

The following (14) tests use SHORT_INTEGER, which is not supported by this compiler.

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	

C45231D requires a macro substitution for any predefined numeric types other than INTEGER, SHORT_INTEGER, LONG_INTEGER, FLOAT, SHORT_FLOAT, and LONG_FLOAT. This compiler does not support any such types.

C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.

C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.

D64005G compiles successfully but does not link in the 64K memory capability of the target.

B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

C36001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.

CA2009C and CA2009F instantiate generic units before the units' bodies are compiled. This compiler requires that such bodies be compiled before the unit is instantiated.

CA3004E, EA3004C, and LA3004A use the INLINE pragma for procedures, which is not supported by this compiler.

CC1221A compiles successfully but does not link in the 64K memory capability of the target.

The following 178 tests are inapplicable because sequential, text, and direct access files are not supported.

CE2102C	CE2102G..H(2)	CE2102K	CE2104A..D(4)
CE2105A..B(2)	CE2106A..B(2)	CE2107A..I(9)	CE2108A..D(4)
CE2109A..C(3)	CE2110A..C(3)	CE2111A..E(5)	CE2111G..H(2)
CE2115A..B(2)	CE2201A..C(3)	EE2201D..E(2)	CE2201F..G(2)
CE2204A..B(2)	CE2208B	CE2210A	CE2401A..C(3)
EE2401D	CE2401E..F(2)	EE2401G	CE2404A
CE2405B	CE2406A	CE2407A	CE2408A
CE2409A	CE2410A	CE2411A	AE3101A
CE3102B	EE3102C	CE3103A	CE3104A
CE3107A	CE3108A..B(2)	CE3109A	CE3110A
CE3111A..E(5)	CE3112A..B(2)	CE3114A..B(2)	CE3115A
CE3203A	CE3301A..C(3)	CE3302A	CE3305A
CE3402A..D(4)	CE3403A..C(3)	CE3403E..F(2)	CE3404A..C(3)
CE3405A..D(4)	CE3406A..D(4)	CE3407A..C(3)	CE3408A..C(3)
CE3409A	CE3409C..F(4)	CE3410A	CE3410C..F(4)
CE3411A	CE3412A	CE3413A	CE3413C
CE3602A..D(4)	CE3603A	CE3604A	CE3605A..E(5)
CE3606A..B(2)	CE3704A..B(2)	CE3704D..F(3)	CE3704M..O(3)
CE3706D	CE3706F	CE3804A..E(5)	CE3804G
CE3804I	CE3804K	CE3804M	CE3805A..B(2)
CE3806A	CE3806D..E(2)	CE3905A..C(3)	CE3905L
CE3906A..C(3)	CE3906E..F(2)		

The following 285 tests require a floating-point accuracy that exceeds the maximum of 9 digits supported by this implementation:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code,

processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into sub-tests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 10 Class B tests.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B33301A	B55A01A	B67001A	B67001C	B67001D
BA1101B2	BA1101B4	BC1109A	BC1109C	BC1109D

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the InterACT Ada 1750A Compiler System was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the InterACT Ada 1750A Compiler System using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a VAX 11/785 operating under VMS, Version 4.5 and a Fairchild F9450/1750A without an operating system. The host and target computers were linked via VAX 64000 Interface Software using the HP 64000 Workstation.

A magnetic tape containing all tests was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized on-site after the magnetic tape was loaded. Tests requiring modifications during the prevalidation testing were not included in their modified form on the magnetic tape. The contents of the magnetic tape were not loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled and linked on the VAX 11/785, and all executable tests were run on the Fairchild F9450/1750A. Object files were linked on the host computer, and executable images were transferred to the target computer via VAX 64000 Interface Software. Results were printed from the host computer, with results being transferred to the host computer via VAX

64000 Interface Software.

The compiler was tested using command scripts provided by InterACT Corporation and reviewed by the validation team. The compiler was tested using all default switch settings.

Tests were compiled, linked, and executed as appropriate using a single host computer and a single target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at InterACT Corporation, New York, New York and was completed on 16 July 1988.

Testing was performed in a mixed batch mode with other on-going processes.

APPENDIX A

DECLARATION OF CONFORMANCE

Appendix A

DECLARATION OF CONFORMANCE

Compiler Implementer: InterACT Corporation
Ada Validation Facility: National Bureau of Standards
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name: InterACT Ada1750A Compiler System Release 3.0
Host Architecture - ISA: VAX11/785 OS&VER #: VMS 4.5
Target Architecture - ISA: Fairchild 9450/ 1750A OS&VER #: bare machine

Derived Compiler Registration

Derived Compiler Name: InterACT Ada1750A Compiler System Release 3.0
Host Architecture - ISA: Any VAX series OS&VER #: 4.3 or greater
Target Architecture - ISA: Any SEAFAC certified 1750A OS&VER #: No OS required

Implementer's Declaration

I, the undersigned, representing InterACT Corp. have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that InterACT Corp. is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

5/2/88

Owner's Declaration

I, the undersigned, representing InterACT Corp. take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their ~~host/target performance are in~~ compliance with the Ada Language Standard ANSI/MIL-STD-1815A. I have reviewed the Validation Summary Report for the compiler(s) and concur with the contents.

5/2/88

This document is part of the Validation Summary Report (VSR), Appendix A, for initial validations and must be submitted for each derived compiler registration during or subsequent to initial validation.

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the InterACT Ada 1750A Compiler System, Release 3.0, are described in the following sections which discuss topics in Appendix F of the Ada Standard. Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is

    type INTEGER is range -32_768 .. 32_767;

    type LONG_INTEGER is range -2_147_483_648 .. 2_147_483_647;

    type FLOAT is digits 6 range -1.0*2.0**127 ..
                                  0.999999*2.0**127;

    type LONG_FLOAT is digits 9 range -1.0*2.0**127 ..
                                  0.999999*2.0**127;

    type DURATION is delta 2**(-14) range -131_072.0 .. 131_071.0;

end STANDARD;
```

Appendix F
Appendix F of the Ada Reference Manual

This appendix describes all implementation-dependent characteristics of the Ada language as implemented by the InterACT Ada 1750A Compiler, including those required in the Appendix F frame of *Ada RM*.

F.1. Predefined Types in Package STANDARD

This section describes the implementation-dependent predefined types declared in the predefined package STANDARD [*Ada RM Annex C*], and the relevant attributes of these types.

Integer Types

Two predefined integer types are implemented, INTEGER and LONG_INTEGER. They have the following attributes:

INTEGER'FIRST	=	-32_768
INTEGER'LAST	=	32_767
INTEGER'SIZE	=	16
LONG_INTEGER'FIRST	=	-2_147_483_648
LONG_INTEGER'LAST	=	2_147_483_647
LONG_INTEGER'SIZE	=	32

Floating Point Types

Two predefined floating point types are implemented, FLOAT and LONG_FLOAT. They have the following attributes:

FLOAT'DIGITS	=	6
FLOAT'EPSILON	=	9.53674316406250E-07
FLOAT'FIRST	=	-1.0 * 2.0**127
FLOAT'LARGE	=	1.93428038904620E+25
FLOAT'LAST	=	0.999999 * 2.0**127
FLOAT'MACHINE_EMAX	=	127
FLOAT'MACHINE_EMIN	=	-128
FLOAT'MACHINE_MANTISSA	=	23

FLOAT'MACHINE_OVERFLOWS	=	TRUE
FLOAT'MACHINE_RADIX	=	2
FLOAT'MACHINE_ROUNDS	=	FALSE
FLOAT'MANTISSA	=	21
FLOAT'SAFE_EMAX	=	127
FLOAT'SAFE_LARGE	=	FLOAT'LAST
FLOAT'SAFE_SMALL	=	0.5 * 2.0**(-127)
FLOAT'SIZE	=	32
LONG_FLOAT'DIGITS	=	9
LONG_FLOAT'EPSILON	=	9.31322574615479E-10
LONG_FLOAT'FIRST	=	-1.0 * 2.0**127
LONG_FLOAT'LARGE	=	2.0**124*(1.0-2.0**(-31))
LONG_FLOAT'LAST	=	.99999999* 2.0**127
LONG_FLOAT'MACHINE_EMAX	=	127
LONG_FLOAT'MACHINE_EMIN	=	- 128
LONG_FLOAT'MACHINE_MANTISSA	=	39
LONG_FLOAT'MACHINE_OVERFLOWS	=	TRUE
LONG_FLOAT'MACHINE_RADIX	=	2
LONG_FLOAT'MACHINE_ROUNDS	=	FALSE
LONG_FLOAT'MANTISSA	=	31
LONG_FLOAT'SAFE_EMAX	=	127
LONG_FLOAT'SAFE_LARGE	=	LONG_FLOAT'LAST
LONG_FLOAT'SAFE_SMALL	=	0.5 * 2**(-127)
LONG_FLOAT'SIZE	=	48

Fixed Point Types

Two kinds of anonymous predefined fixed point types are implemented: *fixed* and *long_fixed*. Note that these names are not defined in package STANDARD, but are used here only for reference.

For objects of *fixed* types, 16 bits are used for the representation of the object. For objects of *long_fixed* types, 32 bits are used for the representation of the object.

For *fixed* and *long_fixed* there is a virtual predefined type for each possible value of *small* [*Ada RM 3.5.9*]. The possible values of *small* are the powers of two that are representable by a LONG_FLOAT value.

The lower and upper bounds of these types are:

lower bound of *fixed* types	=	-32768 * *small*
upper bound of *fixed* types	=	32767 * *small*
lower bound of *long_fixed* types	=	-2_147_483_648 * *small*
upper bound of *long_fixed* types	=	2_147_483_647 * *small*

A declared fixed point type is represented as that predefined *fixed* or *long_fixed* type which has the largest value of *small* not greater than the declared delta, and which has the smallest range that includes the declared range constraint.

Any fixed point type T has the following attributes:

T'MACHINE_OVERFLOWS	=	TRUE
T'MACHINE_ROUNDS	=	FALSE

Type DURATION

DURATION'AFT	=	5
DURATION'DELTA	=	DURATION'SMALL
DURATION'FIRST	=	-131_072.0
DURATION'FORE	=	7
DURATION'LARGE	=	1.31071999938965E05
DURATION'LAST	=	131_071.0
DURATION'MANTISSA	=	31
DURATION'SAFE_LARGE	=	DURATION'LARGE
DURATION'SAFE_SMALL	=	DURATION'SMALL
DURATION'SIZE	=	32
DURATION'SMALL	=	6.10351562500000E-05 = 2**(-14)

F.2. Pragmas

This section lists all language-defined pragmas and any restrictions on their use and effect as compared to the definitions given in *Ada RM.*

Pragma CONTROLLED

This pragma has no effect, as no automatic storage reclamation is performed before the point allowed by the pragma.

Pragma ELABORATE

As in *Ada RM.*

Pragma INLINE

This pragma causes inline expansion to be performed, except in the following cases:

1. The whole body of the subprogram for which inline expansion is wanted has not been seen. This ensures that recursive procedures cannot be inline expanded.

2. The subprogram call appears in an expression on which conformance checks may be applied, i.e., in a subprogram specification, in a discriminant part, or in a formal part of an entry declaration or accept statement.

3. The subprogram is an instantiation of the predefined generic subprograms UNCHECKED_CONVERSION or UNCHECKED_DEALLOCATION. Calls to such subprograms are expanded inline by the compiler automatically.

4. The subprogram is declared in a generic unit. The body of that generic unit is compiled as a secondary unit in the same compilation as a unit containing a call to (an instance of) the subprogram.

5. The subprogram is declared by a renaming declaration.

6. The subprogram is passed as a generic actual parameter.

A warning is given if inline expansion is not achieved.

Note that the primary optimizing effect of this implementation of inline expansion is the elimination or reduction of parameter passing code, rather the reduction of basic subprogram call overhead.

Pragma INTERFACE

This pragma is supported for the language names defined by the enumerated type INTERFACE_LANGUAGE in package SYSTEM. Languages other than BIF support Ada calls to subprograms whose bodies are written in that language. Language BIF (for "built-in function") supports inline insertion of assembly language macro invocations; the macros themselves may consist of executions of 1750A hardware built-in functions, or of any sequence of 1750A instructions. Thus, pragma INTERFACE (BIF) serves as an alternative to machine code insertions.

Language ASSEMBLY

For pragma INTERFACE (ASSEMBLY), the compiler generates a call to the name of the subprogram. The subprogram name must not exceed 31 characters in length. Parameters and results, if any, are passed in the same fashion as for a normal Ada call (see Appendix P).

Assembly subprogram bodies are not elaborated at runtime, and no runtime elaboration check is made when such subprograms are called.

Assembly subprogram bodies may in turn call Ada program units, but must obey all Ada calling and environmental conventions in doing so. Furthermore, Ada dependencies (in the form of context clauses) on the called program units must exist. That is, merely calling Ada program units from an assembly subprogram body will not make those program units visible to the Ada Linker.

A pragma INTERFACE (ASSEMBLY) subprogram may be used as a main program. In this case, the procedure specification for the main program must contain context clauses that will (transitively) name all Ada program units.

If an Ada subprogram declared with pragma INTERFACE (ASSEMBLY) is a library unit, the assembled subprogram body object code module must be put into the program library via the Ada Library Injection Tool (see Chapter 7). The Ada Linker will then automatically include the object code of the body in a link, as it would the object code of a normal Ada body.

If the Ada subprogram is not a library unit, the assembled subprogram body object code module cannot be put into the program library. In this case, the user must direct the Ada Linker to the directory containing the object code module (via the /user_rts qualifier, see Section 5.1), so that the 1750A Linker can find it.

Language BIF

For pragma INTERFACE (BIF), the compiler generates an inline macro invocation that is the name of the subprogram. The subprogram name must not exceed 31 characters in length. Subprogram parameters and results, if any, are passed in the same fashion as for a normal Ada call (see Appendix P), except that the macro invocation replaces the call. No macro arguments are passed on the invocation.

A macro file must exist at the time of the compile containing a macro definition with the same name as the subprogram. This macro file should have a file name that is the same as the subprogram, and a file type of mac. The file should either be located in the current default directory, or be defined by one of two logical names: maclib, or the macro name itself. (See the *InterACT 1750A Assembler and Linker User's Manual* for a full explanation.)

Languages JOVIAL and FORTRAN

These languages may also be specified for pragma INTERFACE, but are equivalent to language ASSEMBLY. The compiler generates calls to such subprograms as if they were Ada subprograms, and does not do any

special data mapping or parameter passing peculiar to the InterACT JOVIAL or FORTRAN compilers.

Pragma LIST

As in *Ada RM*.

Pragma MEMORY_SIZE

This pragma has no effect. See pragma SYSTEM_NAME.

Pragma OPTIMIZE

This pragma has no effect.

Pragma PACK

This pragma is accepted for array types whose component type is an integer or enumeration type that may be represented in 16 bits or less. The pragma has the effect that in allocating storage for an object of the array type, the object components are each packed into the next largest 2^n bits needed to contain a value of the component type. For example, integer components with the range constraint -8 .. 7 are packed into 4 bits; boolean components are packed into one bit.

This pragma is also accepted for record types but has no effect. Record representation clauses may be used to "pack" components of a record into any desired number of bits; see Section F.6.

Pragma PAGE

As in *Ada RM*.

Pragma PRIORITY

As in *Ada RM*. See the *Ada 1750A Runtime Executive Programmer's Guide* for how a default priority may be set.

Pragma SHARED

This pragma has no effect, in terms of the compiler (and a warning message is issued). However, based on the current method of code generation, the effect of pragma SHARED is automatically achieved for all scalar and access objects.

Pragma STORAGE_UNIT

This pragma has no effect. See pragma SYSTEM_NAME.

Pragma SUPPRESS

Only the "identifier" argument, which identifies the type of check to be omitted, is allowed. The "[ON =>] name" argument, which isolates the check omission to a specific object, type, or subprogram, is not supported.

Pragma SUPPRESS with DIVISION_CHECK and OVERFLOW_CHECK has no effect. However, through runtime executive customizations (see the *Ada 1750A Runtime Executive Programmer's Guide*), the overflow interrupts that are used to implement those checks may be masked. Pragma SUPPRESS with all other checks results in the corresponding checking code not being generated.

Pragma SYSTEM_NAME

This pragma has no effect. The only possible SYSTEM_NAME is MIL_STD_1750A. The compilation of pragma MEMORY_SIZE, pragma STORAGE_UNIT, or this pragma does not cause an implicit recompilation of package SYSTEM.

F.3. Implementation-dependent Pragmas

F.3.1. Program Library Basis Pragmas

Certain pragmas defined by this Compiler System apply to Ada programs as a whole, rather than to individual compilation units or declarative regions. These pragmas are NO_DYNAMIC_OBJECTS_OR_VALUES_USED, NO_DYNAMIC_MULTIDIMENSIONAL_ARRAYS_USED, and SET_MACHINE_OVERFLOWS_FALSE_FOR_ANONYMOUS_FIXED.

These pragmas apply on a program library wide basis, and thus apply to any and all programs compiled and linked from a given program library. The meanings of these pragmas is described in the subsections below; the way in which these pragmas are specified is described in this subsection.

These pragmas may only be specified within the implementation-defined library unit LIBRARY_PRAGMAS, which in turn may only be compiled into the Compiler System predefined library. If either of these restrictions are not honored, the pragmas have no effect.

The contents of this library unit when delivered are

```
package LIBRARY_PRAGMAS is

  NO_DYNAMIC_OBJECTS_OR_VALUES_USED : constant BOOLEAN := FALSE;

  NO_DYNAMIC_MULTIDIMENSIONAL_ARRAYS_USED : constant BOOLEAN := FALSE;

  SET_MACHINE_OVERFLOWS_FALSE_FOR_ANONYMOUS_FIXED : constant BOOLEAN := FALSE;

end LIBRARY_PRAGMAS;
```

In order to specify any or all of the pragmas, the source for this package is modified to include the pragmas after the constant declarations (the source file is defined by the logical name **actada_library_pragmas**). For example,

```
package LIBRARY_PRAGMAS is

  NO_DYNAMIC_OBJECTS_OR_VALUES_USED : constant BOOLEAN := FALSE;

  NO_DYNAMIC_MULTIDIMENSIONAL_ARRAYS_USED : constant BOOLEAN := FALSE;

  SET_MACHINE_OVERFLOWS_FALSE_FOR_ANONYMOUS_FIXED : constant BOOLEAN := FALSE;

  pragma NO_DYNAMIC_OBJECTS_OR_VALUES_USED;

  pragma SET_MACHINE_OVERFLOWS_FALSE_FOR_ANONYMOUS_FIXED;

end LIBRARY_PRAGMAS;
```

This modified source is then compiled into the predefined library. To do this, unit LIBRARY_PRAGMAS must first be unlocked via Ada PLU (see Chapter 3).

In addition to the effects described in the subsections below, the pragmas have the effect of changing the initialization value to TRUE for the corresponding constant objects.

If unit LIBRARY_PRAGMAS is modified and compiled by the user, *it must be compiled before any other user compilation unit*. If it is not, the program will be erroneous.

Note that while these pragmas apply to an entire program library, it is possible to create more than one program library (via the Ada PLU command create/root; see Chapter 3), with each library having these pragmas specified or not according to user desire.

F.3.2. Pragma NO_DYNAMIC_OBJECTS_OR_VALUES_USED

This pragma works on a program library basis. See the subsection at the beginning of this section for how such pragmas are used.

Use of this pragma informs the compiler that all created objects and all computed values have statically known sizes. The language usages that do *not* meet this assertion are

- T'IMAGE for integer types
- arrays objects or values of (sub)types with non-static index constraints, or with component subtypes with non-static index constraints
- array aggregates of an unconstrained type
- catenations (even with statically sized operands)
- collections with non-static sizes

Programs that violate the assertion of this pragma are erroneous.

The effect of this pragma is to use a different, and more efficient, set of compiler protocols for runtime stack organization and register usage. These variant protocols are described in Appendix P.

F.3.3. Pragma NO_DYNAMIC_MULTIDIMENSIONAL_ARRAYS_USED

This pragma works on a program library basis. See the subsection at the beginning of this section for how such pragmas are used.

Use of this pragma informs the compiler that all declarations of multidimensional array types or objects have static index constraints [*Ada RM 4.9 (11)*], and that the component subtypes of such arrays, if arrays themselves, also have static index constraints. That is, all multidimensional arrays have statically known size. Programs that violate the assertion of this pragma are erroneous.

The effect of this pragma is to use a special technique, known as *bias vectors*, in the generated code for the calculation of array indexed component offsets for multi-dimensional arrays. This technique involves building a data structure that contains some precomputed offsets, and then indexing into that structure. The major advantage of this technique is that few or no multiplication operations need be generated.

The bias vector data structures are allocated as part of elaboration of the constrained array subtype declaration (or object declaration that implicitly declares such a subtype).

Bias vectors are not used if the array index base type is LONG_INTEGER or if pragma PACK applies to the array.

F.3.4. Pragmas ESTABLISH_OPTIMIZED_REFERENCE and ASSUME_OPTIMIZED_REFERENCE

These pragmas are used to direct the compiler to generate code that more efficiently references objects in a package. This efficiency is achieved by using a base register to address the package objects.

Pragma ESTABLISH_OPTIMIZED_REFERENCE instructs the compiler to load a base register with the beginning address of the objects in the designated package, and to access such objects using the base register. The pragma has the form

pragma ESTABLISH_OPTIMIZED_REFERENCE (*package*_name);

The pragma may appear anywhere within a program unit; the load and subsequent usage of the base register will begin at the point of the pragma appearance. The pragma applies only to the program unit it appears in; it does not apply to program units nested within that unit.

Pragma ASSUME_OPTIMIZED_REFERENCE instructs the compiler to assume that the designated package's beginning address has been loaded into a base register, and to access such objects using the base register. The pragma has the form

pragma ASSUME_OPTIMIZED_REFERENCE (*package*_name);

The pragma should appear at the beginning of the declarative part of a program unit. The pragma applies only to the program unit it appears in; it does not apply to program units nested within that unit. It is not necessary to use this pragma after an instance of pragma ESTABLISH_OPTIMIZED_REFERENCE; rather, it must be used in program units that are called from the unit that contains the pragma ESTABLISH_OPTIMIZED_REFERENCE. If there are intervening (in terms of calls) units between the unit containing pragma ESTABLISH_OPTIMIZED_REFERENCE and the unit desiring to use pragma ASSUME_OPTIMIZED_REFERENCE, then those intervening units must also use pragma ASSUME_OPTIMIZED_REFERENCE.

The pragmas apply only to packages that are library units. Only the objects in the specification part of the package, and within base register range of the package beginning, are accessed by base register.

Only one base register is used by these pragmas, that being register 12. Thus, the pragmas can be in effect for only one package at any given time during execution.

An example of the use of these pragmas:

```
package GLOBAL_VARS is
   ...
end GLOBAL_VARS;

with GLOBAL_VARS;  use GLOBAL_VARS;
procedure P is
   pragma ESTABLISH_OPTIMIZED_REFERENCE (GLOBAL_VARS);
   ...
```

```
      procedure INNER is
         pragma ASSUME_OPTIMIZED_REFERENCE (GLOBAL_VARS);
      begin
         ...
      end INNER;
   begin
      ...
      INNER;
      ...
   end P;
```

F.3.5. Pragma INTERFACE_SPELLING

This pragma is used to define the external name of a subprogram written in another language, if that external name is different from the subprogram name (if the names are the same, the pragma is not needed). The pragma has the form

pragma INTERFACE_SPELLING (*subprogram*_name, *external_name*_string_literal);

The pragma should appear after the pragma INTERFACE for the subprogram. This pragma is useful in cases where the desired external name contains characters that are not valid in Ada identifiers. For example,

```
procedure CONNECT_BUS (SIGNAL : INTEGER);
pragma INTERFACE (ASSEMBLY, CONNECT_BUS);
pragma INTERFACE_SPELLING (CONNECT_BUS, "$CONNECT.BUS");
```

F.3.6. Pragma SET_MACHINE_OVERFLOWS_FALSE_FOR_ANONYMOUS_FIXED

This pragma works on a program library basis. See the subsection at the beginning of this section for how such pragmas are used.

The effect of this pragma is that any fixed point type T of anonymous predefined *fixed* type (i.e., represented in 16 bits) has the attribute

T'MACHINE_OVERFLOWS = FALSE

such that NUMERIC_ERROR is not raised in overflow situations [*Ada RM 4.5.7* (*7*)].

The result of operations in overflow situations is either the lower or upper bound of the "virtual" predefined type for T ([*Ada RM 3.5.9* (*10*)], this document Section F.1), depending on the direction of overflow. These bounds are -32_768 * T'SMALL and 32_767 * T'SMALL respectively. These bounds will equal T'FIRST and T'LAST if the range constraint for T is so declared.

Note that this implementation of fixed point types relies on the 1750A fixed point overflow interrupt being enabled and not masked; any user exit or customization routines in the Ada runtime executive must not do differently.

F.3.7. Pragma SUBPROGRAM_SPELLING

This pragma is used to define the external name of an Ada subprogram. Normally such names are compiler-generated, based on the program library unit number. The pragma has the form

pragma SUBPROGRAM_SPELLING (*subprogram*_name [,*external_name*_string_literal]);

The pragma is allowed wherever a pragma INTERFACE would be allowed for the subprogram. If the second argument is omitted, the subprogram name is used as the external name.

This pragma is useful in cases where the subprogram is to be referenced from another language.

F.4. Implementation-dependent Attributes

None are defined.

F.5. Package SYSTEM

The specification of package SYSTEM is:

```
package SYSTEM is

  type ADDRESS              is new INTEGER;
  ADDRESS_NULL              : constant ADDRESS := 0;
  ADDRESS_ZERO              : constant ADDRESS := 0;

  type NAME                 is (MIL_STD_1750A);

  SYSTEM_NAME               : constant NAME := MIL_STD_1750A;

  STORAGE_UNIT              : constant := 16;
  MEMORY_SIZE               : constant := 64 * 1024;

  MIN_INT                   : constant := -2_147_483_647-1;
  MAX_INT                   : constant := 2_147_483_647;
  MAX_DIGITS                : constant := 9;
  MAX_MANTISSA              : constant := 31;
  FINE_DELTA                : constant := 1.0 / 2.0 ** MAX_MANTISSA;
  TICK                      : constant := 0.000_010;

  subtype PRIORITY          is INTEGER range 0..255;

  type INTERFACE_LANGUAGE   is (ASSEMBLY, BIF, JOVIAL, FORTRAN);

end SYSTEM;
```

F.6. Representation Clauses

In general, no representation clauses may be given for a derived type. The representation clauses that are accepted for non-derived types are described by the following:

Length Clause

The compiler accepts three kinds of length clauses, specifying the number of storage units to be reserved for a collection (attribute designator STORAGE_SIZE), the number of storage units to be reserved for an activation of a task (STORAGE_SIZE), or the *small* for a fixed point type (SMALL). Length clauses specifying object size for a type (SIZE) are not allowed.

Enumeration Representation Clause

Enumeration representation clauses may only specify representations in the range of the predefined type INTEGER.

Record Representation Clause

In terms of allowable component clauses, record components fall into three classes:

- integer and enumeration types that may be represented in 16 bits or less;
- statically-bounded arrays or records composed solely of the above;
- all others.

Components of the "16-bit integer/enumeration" class may be given a component clause that specifies a storage place at any bit offset, and for any number of bits, as long as the storage place is large enough to contain the component and does not cross a word boundary.

Components of the "array/record of 16-bit integer/enumeration" class may be given a component clause that specifies a storage place at any bit offset, if the size of the array/record is less than a word, or at a word offset otherwise, and for any number of bits, as long as the storage place is large enough to contain the component and none of the individual integer/enumeration elements of the array/record cross a word boundary.

Components of the "all others" class may only be given component clauses that specify a storage place at a word offset, and for the number of bits normally allocated for objects of the underlying base type.

Components that do not have component clauses are allocated in storage places beginning at the next word boundary following the storage place of the last component in the record that has a component clause.

Alignment clauses are not allowed.

F.7. Implementation-dependent Names for Implementation-dependent Components

None are defined.

F.8. Address Clauses

Address clauses are supported for objects that are not constants, for subprogram units, and for interrupt entries. Address clauses are not supported for package or task units, and in general are not supported for constant objects.

Address Clause for Objects

Address clauses for objects must be static expressions of type ADDRESS in package SYSTEM. Address clauses for objects do not cause the object to be placed at that address, but do ensure that all references to the object in the generated code are to that address. Thus, it is the user's responsibility to reserve space for the object at that address, via 1750A Linker control statements.

Type ADDRESS is a 16-bit signed integer. Thus, addresses in the memory range 16#8000#..16#FFFF# (i.e., the upper half of 1750A memory) must be supplied as negative numbers, since the positive (unsigned) interpretations of those addresses are greater than ADDRESS'LAST. To illustrate:

```
X : INTEGER;
for X use at 16#7FFF#;   -- legal

Y : INTEGER;
for Y use at 16#FFFF#;   -- illegal

Y : INTEGER;
for Y use at -1;         -- legal, equivalent to unsigned 16#FFFF#
```

The hexadecimal address can be retained, and user computation of the negative equivalent avoided, by use of the following construct:

```
ADDR_FFFF : constant := 16#FFFF#-65536;

Y : INTEGER;
for Y use at ADDR_FFFF;
```

Address Clause for Subprogram Units

Address clauses for subprograms must be static expressions of type ADDRESS in package SYSTEM. The code of the subprogram body will be placed at that address. There is no need for the user to reserve space for the subprogram code via the 1750A Linker, as in the case for address clauses for objects.

Address Clause for Interrupt Entries

Address clauses for interrupt entries do not use type SYSTEM.ADDRESS; rather, the address clause must be a static integer expression in the range 0..15, naming the corresponding 1750A interrupt.

The following restrictions apply to interrupt entries. The corresponding accept statement must have no formal parameters and must not be part of a select statement. Direct calls to the entry are not allowed. If any

exception can be raised from within the accept statement, the accept statement must include an exception handler. The accept statement cannot include tasking or delay statements.

When the accept statement is encountered, the task is suspended. If the specified interrupt occurs, execution of the accept statement begins. When control reaches end of the accept statement, the special interrupt entry processing ends, and the task continues normal execution. Control must again return to the point where the accept statement is encountered in order for the task to be suspended again, awaiting the interrupt.

There are many more details of how interrupt entries interact with the 1750A machine state and with the Runtime Executive. For these details, see the *Ada 1750A Runtime Executive Programmer's Guide*.

F.9. Unchecked Conversion

Unchecked conversion is only allowed between values of the same size. In addition, if UNCHECKED_CONVERSION is instantiated with an array type, that type must be statically constrained. Note also that calls to UNCHECKED_CONVERSION-instantiated functions are always generated as inline calls by the compiler.

F.10. Input-Output

The predefined library generic packages and packages SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO are supplied. However, file input-output is not supported except for the standard output file. Any attempt to create or open a file will result in USE_ERROR being raised, as will any attempt to perform operations upon the standard input file.

TEXT_IO output operations to the standard output file are implemented as output to some visible device for a given implementation of MIL-STD-1750A. Depending on the implementation, this may be a console, a workstation disk drive, simulator output, etc.

The range of the type COUNT defined in TEXT_IO is 0 .. LONG_INTEGER'LAST.

The predefined library package LOW_LEVEL_IO is empty.

In addition to the predefined library units, a package STRING_OUTPUT is also included in the predefined library. This package supplies a very small subset of TEXT_IO operations to the standard output file. The specification is:

```
package STRING_OUTPUT is

  procedure PUT (ITEM : in STRING);

  procedure PUT_LINE (ITEM : in STRING);

  procedure NEW_LINE;

end STRING_OUTPUT;
```

By using the 'IMAGE attribute function for integer and enumeration types, a fair amount of output can be done using this package instead of TEXT_IO. The advantage of this is that STRING_OUTPUT is smaller than TEXT_IO in terms of object code size, and faster in terms of execution speed.

F.11. Other Chapter 13 Areas

The following language features, defined in [*Ada RM 13*], are supported by the compiler:

- representation attributes [*13.7.2*, *13.7.3*]
- unchecked storage deallocation [*13.10.1*]

Note that calls to UNCHECKED_DEALLOCATION-instantiated procedures are always generated as inline calls by the compiler.

Change of representation [*13.6*] and machine code insertions [*13.8*] are not supported by the compiler. Note that pramga INTERFACE (BIF) may be used as an alternative to machine code insertions.

F.12. Miscellaneous Implementation-dependent Characteristics

Uninitialized Variables

There is no check to detect the use of uninitialized variables. The effect of a program that refers to the value of an uninitialized variable is undefined. A cross-reference listing may be of use in finding such variables.

F.13. Compiler System Capacity Limitations

The following capacity limitations apply to Ada programs in the Compiler System:

- the space available for the constants of a compilation unit is 32K words;
- the space available for the static data of a compilation unit is 32K words;
- any single object can not exceed 32K words;
- the space available for the objects local to a subprogram or block is 32K words;
- the names of all identifiers, including compilation units, may not exceed the number of characters specified by the INPUT_LINELENGTH component in the compiler configuration file (see Section 4.1.4);
- the physical size of a sublibrary may not exceed 16384 VAX/VMS blocks.

The above limitations are all diagnosed by the compiler. Most may be circumvented straightforwardly by using separate compilation facilities or by creating new sublibraries.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	<1..125 => 'A', 126 => '1'>
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	<1..125 => 'A', 126 => '2'>
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	<1..62 => 'A', 62 => '3', 64..126 => 'A'>
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	<1..62 => 'A', 63 => '4', 64..126 => 'A'>
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	<1..123 => '0', 124..125 => '298'>
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	<1..120 => '0', 121..126 => '69.0E1'>

$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	<1 => '"', 2..64 => 'A', 65 => '"'>
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	<1 => '"', 2..63 => 'A', 64 => '1', 65 => '"'>
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	106
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_647
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	35
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	BAD_FILENAME_1
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	BAD_FILENAME_2
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	131_072.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	131_072.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	ILLEGAL_FILE_NAME_1

$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	ILLEGAL_FILE_NAME_2
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32_768
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	32_767
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	32_768
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-131_073.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-131_073.0
$MAX_DIGITS Maximum digits supported for floating-point types.	9
$MAX_IN_LEN Maximum input line length permitted by the implementation.	126
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	<1..2 => '2:', 3..123 => '0', 124..126 => '11:'>

$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	<1..3 => '16:', 4..122 => '0', 123..126 => 'F.E:'>
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	<1 => '"', 2..125 => 'A', 126 => '"'>
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_ INT.	-2147483648
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	No_Such_Type
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 28 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

B28003A: A basic declaration (line 36) wrongly follows a later declaration.

E28005C: This test requires that 'PRAGMA LIST (ON);' not appear in a listing that has been suspended by a previous "pragma LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the ARG.

C34004A: The expression in line 168 wrongly yields a value outside of the range of the target type T, raising CONSTRAINT_ERROR.

C35502P: Equality operators in lines 62 & 69 should be inequality operators.

A35902C: Line 17's assignment of the nominal upper bound of a fixed-point type to an object of that type raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may in fact raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

C35A03E, & R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard doesn't support this assumption.

C37213H: The subtype declaration of SCONS in line 100 is wrongly expected to raise an exception when elaborated.

C37213J: The aggregate in line 451 wrongly raises CONSTRAINT_ERROR.

C37215C, E, G, H: Various discriminant constraints are wrongly expected to be incompatible with type CONS.

C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

C41402A: 'STORAGE_SIZE is wrongly applied to an object of an access type.

C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

C45614C: REPORT.IDENT_INT has an argument of the wrong type (LONG_INTEGER).

E66001D: Wrongly allows either the acceptance or rejection of a parameterless function with the same identifier as an enumeration literal; the function must be rejected (see Commentary AI-00330).

A74106C, C85018B, C87B04B, CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur re lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively (and possibly elsewhere).

BC3105A: Lines 159..168 are wrongly expected to be illegal; they are legal.

AD1A01A: The declaration of subtype INT3 raises CONSTRAINT_ERROR for implementations that select INT'SIZE to be 16 or greater.

CE2401H: The record aggregates in lines 105 & 117 contain the wrong values.

CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.